# Improving X10 Program Performances by Clock Removal

Paul Feautrier, Eric Violard, Alain Ketterlin

# Improving X10 Program Performances by Clock Removal

Paul Feautrier[1], Éric Violard[2], and Alain Ketterlin[2]

[1] INRIA, UCBL, CNRS & Ecole Normale Supérieure de Lyon, LIP, Compsys
[2] INRIA & Université de Strasbourg

**Abstract.** X10 is a promising recent parallel language designed specifically to address the challenges of productively programming a wide variety of target platforms. The sequential core of X10 is an object-oriented language in the Java family. This core is augmented by a few parallel constructs that create *activities* as a generalization of the well known fork/join model. Clocks are a generalization of the familiar barriers. Synchronization on a clock is specified by the `advance()` method call. Activities that execute *advances* stall until all existent activities have done the same, and then are released at the same (logical) time.
This naturally raises the following question: are clocks strictly necessary for X10 programs? Surprisingly enough, the answer is no, at least for sufficiently regular programs. One assigns a date to each operation, denoting the number of advances that the activity has executed before the operation. Operations with the same date constitute a *front*, fronts are executed sequentially in order of increasing dates, while operations in a front are executed in parallel if possible. Depending on the nature of the program, this may entail some overhead, which can be reduced to zero for polyhedral programs. We show by experiments that, at least for the current X10 runtime, this transformation usually improves the performance of our benchmarks. Besides its theoretical interest, this transformation may be of interest for simplifying a compiler or runtime library.

## 1 Introduction

Due to physical limitations, today computers all have explicit parallelism. This is true over the whole power spectrum, from embedded systems to high performance number crunchers, in which millions of cores must contribute to a common task. Efficient programming of such architectures is one of the most important challenge of the next decade. Among the many solutions which have been proposed – parallel programming libraries, domain specific languages, automatic parallelization – one of the most interesting is the use of parallel programming languages: languages in which parallel constructs are first class citizens, on a par with standard control constructs like the sequence or the loop. This approach has two advantages. Firstly, it hides the intricate details of parallel programming at the hardware or operating system level. Second, and most importantly, the programmer can express the problem inherent parallelism. Such parallelism might be difficult to infer from a sequential implementation.

The recent years have seen the creation of many such languages, among which Titanium [1], Chapel [2], Co-Array Fortran [3], UPC [4], Habanero Java [5]. This paper deals with X10[3] which is being developed at IBM Research. However, we believe that our techniques – if not our results – can be adapted without difficulties to other languages. Basically, parallelism is expressed by syntactic constructions, `async/finish` in X10 or `cobegin/coend` in Chapel. For some algorithms, it is necessary to restrict temporarily the degree of parallelism, using synchronization objects, called clocks in X10 or phasers in Habanero Java. These primitives are somewhat redundant, and may be used interchangeably in some circumstances. The aim of this paper is to explore these redundancies for X10, and to evaluate their impact on program performance.

Our key contributions are:

- we give a general scheme for clock elimination, which applies only to static control programs,
- we show that this scheme is correct and does not lose parallelism,
- for polyhedral programs, the control overhead of the target program can be reduced or even eliminated by loop transformations,
- experiments show that for the latest version of the X10 compiler and runtime, the proposed transformation improves the running time for fine grain parallel programs.

The rest of the paper is structured as follows. We will first give as much information on X10 as necessary to understand our approach. We will then define the polyhedral subset of X10. While our approach is not limited to this subset, it gives the best results in the case of polyhedral programs. results.

### 1.1 The X10 Language

**The Base Language** X10 is an object oriented language of the Java family. It has classes and methods, assignments and method invocation, and the usual control constructs: conditionals and loops. Dealing with method invocation necessitates interprocedural analysis, and is beyond the scope of this paper. The exact shape of assignments is irrelevant in this work.

X10 has two kind of loops: the ordinary Java loop:

```
for(<initialization>; <tests>; <increment>) S
```

and an enumerator loop:

```
for(x in <range>) S
```

where the type of the counter `x` is inferred from the type of the range.

[3] x10.sourceforge.net/documentation/languagespec/x10-latest.pdf

**Concurrency** Concurrency is expressed in X10 by two constructs, `async S` and `finish S`, where `S` is an arbitrary statement or statement block. Such constructs can be arbitrarily nested, except that the whole program is always embedded in an implicit or explicit `finish`, which creates the main activity. The effect of `async S` is to create a new *activity* or lightweight thread, which executes `S` in parallel with the rest of the program. The effect of `finish S` is to launch the execution of `S`, then to wait until all activities which were created inside `S` have terminated.

X10 also allows the distribution of work on a set of logical places (typically, various nodes of a compute cluster), in a way that is transparent to the organization of activities. This aspect is orthogonal to the work explained in this paper, and will not be further evoked.

**Synchronization** In some cases, it may be necessary to synchronize several parallel activities. This can be achieved using *clocks*. Clocks are created by `clocked finish` constructs. Activities are *registered* to the clock associated to the innermost enclosing clocked finish if created by a `clocked async` construct. An activity deregisters itself from its associated clock when it terminates. Synchronization occurs when an activity executes the `advance` primitive. This activity is held until all registered activities have executed an `advance`, at which time all registered activities are released.

Clocks can be seen as generalization of the classical barriers. The main differences are that activities may be distributed among several clocks which work independently, and that this distribution is dynamic as it can change when an activity is created or terminated. Refer to Figure 1 for a sample X10 program.

Intuitively, it should be clear that an unclocked finish or a set of advances can be used interchangeably. In both cases, several activities are stalled until all of them have reached a synchronization point. If a clock is used, then all clocked activities are released for further processing, while in the case of a finish, they are destroyed. The aim of this paper is to explore this analogy, both from the point of view of expressiveness and from the point of view of performance.

### 1.2 The Polyhedral Subset of X10

In general, analysis of arbitrary programs in a high level language like X10 is difficult, due to the presence of dynamic constructs like while loops, tests, and method invocation. Hence, many authors [6] have defined the so-called polyhedral model, in which many analyzes can be done at compile time. The polyhedral subset of X10 has been defined in [7]. The present section is a summary of this work. An X10 program is in the polyhedral model if its data structures are arrays and its control structures are loops. An enumerator loop is polyhedral if the range is an integer interval. The bounds of the range and the array subscripts must be affine functions of surrounding loops counters and integer parameters.

If these conditions are met, one can define statement instances, iteration domains, and an order of execution or happens-before relation. Statement in-

stances are named by position vectors, which are deduced from abstract syntax trees (AST).

Consider the following example and its AST:

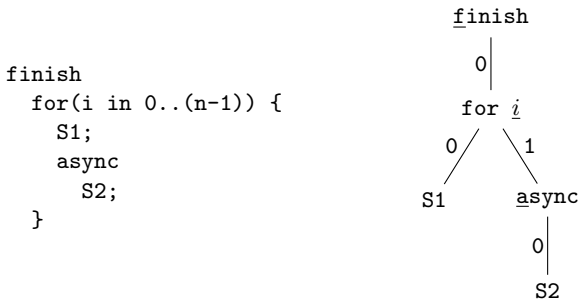


**Fig. 1.** A Sample Program

The position vector for an elementary statement $S$ is obtained by following the unique path in the AST from the root to $S$. In the example, the position vector of $S_1$ is $[f, 0, i, 0]$ and that of $S_2$ is $[f, 0, i, 1, a, 0]$, where $f$ stands for `finish` and $a$ stands for `async`. Let $x$ and $y$ be two position vectors, and let us write $x \prec y$ for "instance $x$ happens before instance $y$". To decide whether $x \prec y$, first expand $x \ll y$, where $\ll$ is the ordinary lexicographic order. Then, remove a term if, after elimination of a common prefix, the first letter one encounter on the left is an $a$. This rule reflects the fact that the only temporal effect of `async S` is to postpone the execution of `S`. The reader may care to check that in the above example, instances of $S_2$ are unordered, while $S_1(i)$ happens before $S_2(i')$ if $i < i'$.

Another construction is necessary for programs that use clocks. The simplest case is that of one-clock programs (or of innermost clocked finishes). One must distinguish the unclocked happens-before relation, for which advances are treated as ordinary statements, and the clocked happens-before, noted $\prec\!\!\prec$. Let $\mathcal{A}$ be the set of advances inside one clocked finish. The advance counter at operation $u$ is defined as:

$$\phi(u) = \mathrm{Card}\{u' \in \mathcal{A} \,|\, u' \prec u\}.$$

When the effect of clocks is taken into account, one can prove that if $\phi(u) < \phi(v)$, then $u$ happens before $v$. As a consequence, the clocked happens-before relation is:

$$u \prec\!\!\prec v \equiv \phi(u) < \phi(v) \vee u \prec v.$$

Since for polyhedral programs $\mathcal{A}$ is a union of disjoint polyhedra, and $u' \prec u$ is a disjunction of affine inequalities, the set $\{u' \in \mathcal{A} \,|\, u' \prec u\}$ is the set of integer points which belong to a union of polyhedra. The cardinal of this set can

be computed in closed form using the theory of Ehrhart polynomial, for which there exists efficient libraries [8].

## 2 A Generic Transformation Strategy

Our goal is to remove clocks from X10 programs. To understand the idea of this transformation, consider Figure 2: the center graph depicts the execution of an imaginary X10 program, where activities are represented by vertical boxes that contain regular instruction executions and clock synchronization operations. These activities "align" on their calls to `advance()`. The code on the left side of the figure is one possible source of this program. The idea of the transformation is to extract "slices" (or phases) across activities, represented by horizontal dashed boxes on the graph. A possible corresponding program appears on the right of the figure: the usage of clocks has been replaced by the barrier ending `finish` blocks. We will prove in the next section that both programs execute the same operations in the same order, except for clocks and the number (and duration) of activities.

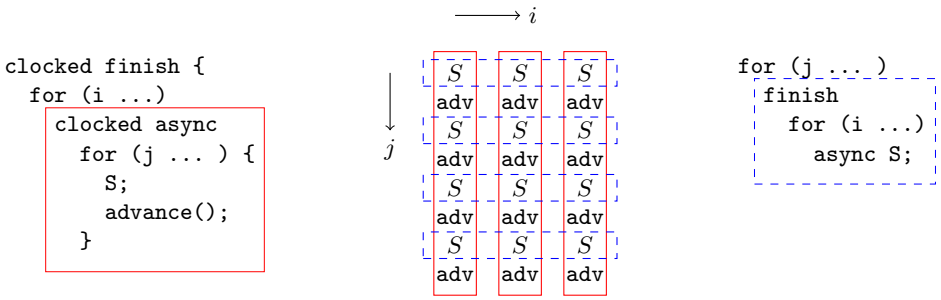


**Fig. 2.** Parallelism and synchronization in X10, with and without clocks

This transformation, can be implemented by a straightforward technique. Starting with a given `clocked finish` block, the result of the transformation can be sketched as follows:

```
for (d ...) // where d is a monotonically increasing phase number
  finish S_d // the original finish block restricted to phase d
```

Writing the transformed program this way assumes that it is possible 1) to determine the number of phases of the program, either statically or dynamically, 2) to execute the given block for a given phase only (the *restriction* of the block to that phase), and 3) to repeatedly execute the original block. The rest of this section explores these three issues.

### 2.1 Motivating Example

Our goal in this section is three-fold. First, it is important to understand what class of programs the transformation can be applied to. Second, the example will help pinpointing potential optimizations. And third, we want to empirically validate our intuition that managing clocks is more expensive than creating activities.

Our working example appears on the left of Figure 3. The `finish` block creates only two activities in addition to the main activity. Each of these execute a loop that does some work (in abstract instructions `S0` and `S1`), and then conditionally synchronizes with the other. A set of input parameters, contained in arrays `a0` and `a1`, drives the control of the program and the synchronization scheme. These parameters make it impossible to statically derive how many phases the program has, and how many executions of `S0` and `S1` are performed in each phase.

```
clocked finish {
  clocked async {
    for (i in 0..(N-1)) {
      S0(i);
      if (a0(i) > 0)
        advance();
    }
  }
  clocked async {
    for (i in 0..(N-1)) {
      S1(i);
      if (a1(i) > 0)
        advance();
    }
  }
}
```

```
1  cont = true;
2  for (d=0 ; cont ; d++) {
3    cont = false;
4    finish {φ := 0;
5      async { φ0 := φ;
6        for (i in 0..(N-1)) {
7          if (d == φ0) S0(i);
8          if (a0(i) > 0)
9            ++ φ0;
10       }
11       if (d<φ0) cont = true; }
12     async { φ1 := φ;
13       for (i in 0..(N-1)) {
14         if (d == φ1) S1(i);
15         if (a1(i) > 0)
16           ++ φ1;
17       }
18       if (d<φ1) cont = true; }
19   if (d<φ) cont = true; }
20 }
```

**Fig. 3.** An example program, before and after transformation

The resulting program appears on the right of Figure 3. The transformation can be broken into four successive steps:

1. The `finish` block is wrapped inside a loop over $d$, whose iterations represent the various phases of the execution (line 2 on Fig. 3). The exit condition is represented with a boolean, named `cont`, whose role is detailed in the fourth phase.

2. Every activity gets its own local "counter" (named $\phi$, $\phi_0$ and $\phi_1$ in the example),[4] initialized at the start of the activity by capturing the value of the parent activity's counter if any (lines 4, 5, and 12). Local counters are maintained by replacing calls to `advance()` by an incrementation (lines 9 and 16).
3. All instructions that have an effect visible outside the `finish` block are guarded (lines 7 and 14), and the guard condition checks whether the value of the local counter matches the currently executed phase (given by $d$).
4. Finally, when any activity reaches its end, the value of the local counter has reached its maximum value for that activity. This maximum value is the index of the last phase for which this activity has work to do. A simple test decides whether the loop on $d$ should continue iterating (lines 11, 18, and 19).[5]

To evaluate the performance impact of the transformation, we still need to give some definition to `S0(i)` and `S1(i)`. In the experiment below, we use some "dummy" code of the form:

```
for (t in 1..T)
    a(i) += garbage(k%4)
```

that is to say, two accesses to arrays plus two arithmetic operations (subject to optimization). The `T` parameter is used to control the amount of work performed by one call to either `S0` and `S1`: on a recent laptop, we have observed that such a loop takes roughly `T` nanoseconds. To run either the original or the modified program, the arrays `a0` and `a1` are filled with randomly generated values with equiprobable signs.

Figure 4 shows the execution times in milliseconds of both versions with $N = 100$ as a function of the parameter `T`. The original version uses clocks to synchronize both activities, whereas the modified version simply repeats the whole `finish` block as many times as necessary (therefore creating many more activities). These curves are surprisingly close to each other. For moderately heavy instruction grain (here between 10 and 100 $\mu$s per call to `S0` or `S1`), it seems that the cost of handling clocks is approximately as high as executing around 50 instances of the block (including the creation of activities). This accomplishes our third goal, and validates our intuition that clocks are expensive.

### 2.2 Applicability and Correctness

There are two main aspects in the generic transformation:

1. guarding the instructions, so as to have them execute during the right phase;
2. maintaining phase numbers ("dates") during each iteration of the loop on $d$.

[4] The local counter of the activity executing the body of `finish`, named $\phi$, is useless here and was left for completeness only.

[5] Activities could be aborted once their local counter is above the value of $d$: this aspect is more or less orthogonal to our goal, and is ignored here.

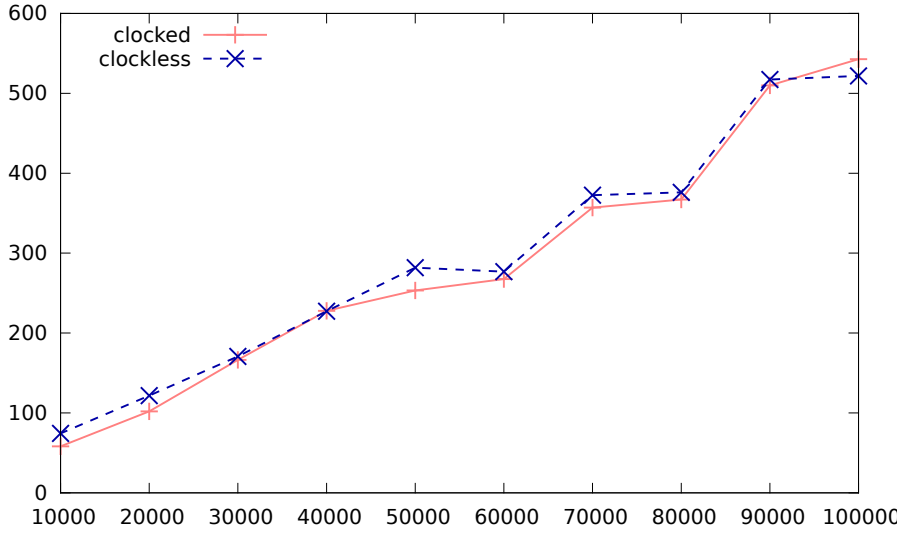


**Fig. 4.** Clocked and clockless execution times (in milliseconds)

Let us for a moment assume that the second aspect is enforced. In that case, it is easy to see that both versions of the program are equivalent. For, if two instructions of the original block are executed during different phases, then they will be executed inside different `finish` blocks in the transformed program. And if they are executed during the same phase, they will still be executed in the right order, since the transformed block is a copy of the original block, and thus faithfully reproduces program order. Therefore, correctness of the transformation is guaranteed if phase numbers can be correctly maintained.

Maintaining correct dates at all times, *i.e.*, during every iteration of the $d$-loop, however, is not possible for all programs. Here is a simple modification of our previous example, where the body of the $i$-loop inside the first `async` becomes:

```
if (a0(i) > 0) {
  a0(i) = -1;
  advance();
}
```

Here, code executed at date $d$ updates a value that will be used at a later date (an iteration $d'$ with $d' > d$). This means that later iterations of the $d$-loop will not be able to maintain phase numbers correctly, leading to an incorrect result.

The general criterion to distinguish programs that can be transformed correctly is the following: every iteration of the $d$-loop must perform exactly the same sequence of advance counter incrementations. To formally capture this notion, let us define *control variables*: a variable that is used in a conditional branch

(including loop back-edges), or to update another control variable (with arrays considered as single variables).[6] Then, a program will be correctly transformed, if the history of each control variable is the same in each iteration of the `d` loop, *i.e.* if no control variable is *live in* at the beginning of the `d` loop. For sequential structured programs, this can be checked by many classical algorithms, including reaching definition analysis and transformation to SSA. These algorithms can be extended to parallel programs: see for instance [9], where a Concurrent Static Single Assignment form is defined for programs with parallel construct similar to those of X10, including `post / wait` synchronization. The results of this analysis are approximate. For polyhedral X10 programs, it is possible to do better, as shown in [7].

### 2.3 Optimization Opportunities

The generic transformation described above uses a very costly strategy: re-executing the original code again and again, inhibiting the execution of almost all instructions at each iteration. Even though this cost seems to be amortized even for moderately heavy computations, the whole structure of the transformed program is unsatisfactory. This section tries to highlight characteristics of the transformed program that may lead to further simplification. The various steps of the intuitive transformation provide important clues on classes of programs where this applies.

The first aspect is about the local counters that have to be maintained to model the "date" inside an activity. In some cases, the date may be available at compile time as a closed form function of loop counters. Or it can be precomputed to avoid repeated incrementations of the local counter. In our example, precomputation would fill two arrays `d0` and `d1`, indexed on `i` and containing the date at iteration `i`. The code of the first activity becomes:

```
for (i in 0..(N-1))
    if (d == d0(i)) S0(i);
```

In other cases, like the ones described in the next section, dates are functions of the enclosing loop counters, and do not need dedicated storage.

The second aspect is very much related to the first, and relates to the upper bound of the enclosing loop. When dates are available, it is immediate to compute or memorize their maximal value (which is the upper bound on `d`) . This also removes the need of a boolean variable and tests at the end of activities.

The third aspect relates to the interplay between statement guards and loop bounds. In our example, we have reached a situation where a loop iterates from 0 to $N-1$, but where only a sub-range of this iteration space leads to actual execution. It is therefore possible to adjust the range of the loop to cover only the relevant sub-range. In our example, the first activity becomes:

[6] In practice, the collection of control-variables can be restricted to programming constructs containing at least one call to `advance()`.

```
while (d0(i0) == d) {
    S0(i0);
    ++ i0;
}
```

where `i0` is a global counter, suitably initialized and preserved across activities.

We have given an informal account on how a program with clocks removed can be further simplified and optimized. The next section describes a class of programs where these optimizations can be systematically applied, and details their implementation.

### 2.4 General Polyhedral Programs

Polyhedral programs with clocks have the property that a date can be computed directly for any instruction, by counting the number of calls to `advance()` performed before an instance of the given instruction. This removes the need to maintain explicit counters, and provides symbolic expressions involving loop counters and symbolic parameters. One can always evaluate the number of integer poins inside a (parametrized) polyhedron, and therefore assign a monotonically increasing rank to any instance of an instruction. Under reasonable assumptions, i.e., that loops have unit steps, such ranks are integer-valued polynomials with rational coefficients [10].

An example program appears on Figure 5, with date expressions placed in comments. Note that the counting happens in two phases: first, the starting date of an activity is computed, and second the date of instructions are computed relative to the activity's starting date.

```
clocked finish
  for (i in 0..(N-1)) {
    clocked async {  // i
      for (j in 0..(M-1)) {
        S0(i,j);     // i+j*(j-1)/2
        for (k in 0..(j-1)) {
          S1(i,j,k); // i+j*(j-1)/2+k
          advance();
    } } }
    advance();
  }
```

```
for (d in 0..(N-2+M*(M-1)/2))
  finish
    for (i in 0..(N-1)) {
      async {
        for (j in 0..(M-1)) {
          if (d == i+j*(j-1)/2)
            S0(i,j);
          for (k in 0..(j-1)) {
            if (d == i+j*(j-1)/2+k)
              S1(i,j,k);
    } } }
    }
```

**Fig. 5.** A polyhedral program with polynomial dates (in comments) on the left, and the result of the transformation, on the right.

The transformation process starts by computing the maximal date at which an instruction of the original `finish` block executes. This can be done by maximizing for each instruction individually, and then taking the maximum of the

results. This maximum, $N - 2 + M(M-1)/2$ in our example, is the upper bound of the loop wrapped around the original block. Then, calls to `advance()` are removed, and guards are placed around statements. The result appears on the right part of Figure 5.

After having inserted guards around statements, the last step is to examine whether the guards have an impact on the bounds of the loops that enclose the statement. Our example illustrates this situation: after transformation, the innermost loop becomes:

```
for (k in 0..(j-1))
    if (d == i+j*(j-1)/2+k)
        S1(i,j,k);
```

A trivial rewriting of the guard shows that even though the loop iterates over a range of values for `k`, the whole loop will actually execute `S1(i,j,k)` at most once. This construct can therefore be replaced by:

```
k = d - i - j*(j-1)/2;
if (0<=k && k<=j-1)
    S1(i,j,k);
```

which tests whether the single value selected by the guard is inside the range of the loop.

Note that we started with a depth three loop nest. Then a new loop level was added around this nest. And finally the deepest level is removed. This is likely to reduce the overhead introduced by the transformation. This optimization may be extended to loops containing several statements (at the same or different dates). However, it applies only when date expressions are linear in the nearest enclosing loop counter, which we think is a very common case. Actually, for this not to be the case, an innermost statement-bearing loop should also contain another loop 1) containing only calls to `advance()`, and 2) with a bound being a function of its parent loop counter. Here is the simplest example of such a construction:

```
for (y in ...) {
    S(...);
    for (z in 0..y)
        advance();
}
```

We think this pathological case and its variations are sufficiently infrequent not to cause real trouble in practice.

Note that after a loop is removed, the statement is still guarded, but with a condition involving inequalities. Therefore, there is no possibility of re-applying the same "iteration space collapsing", but nothing says that the new guard may not imply bound adjustments on the enclosing loops. The next section shows an example of such chained loop adjustments.

### 2.5 Polyhedral Programs with Affine Dates

We have seen that when polynomial dates are available, the resulting program can be optimized by combining guards and loop bounds. However, dealing with polynomials of high degree is difficult and may restricts how far optimizations can go. It is therefore interesting to consider the particular case of affine dates. In that case, all obstacles to optimization are lifted, and one can hope to be able to optimize the transformed program up to the point where it has the same complexity as the original program.[7]

Whenever the original program induces dates that are all affine forms in the enclosing loop counters (and parameters), we are guaranteed that the deepest loop level can be removed. In fact, this last level of loop contains only guarded statements, and the guards are of the form $d = \alpha_0 i_0 + \ldots + \alpha_n i_n$, where $i_0, \ldots, i_n$ are the counters of the enclosing loops. Such a guard always determines at most one value of the counter of the nearest loop. This property appears in the program in Figure 6. The left part shows the original `clocked finish` block (dates appear in comments), whereas the right part shows the mechanically transformed program. Since dates are affine, one can immediately apply the "iteration space collapsing" optimization mentioned in the previous section. The first loop on `j` then becomes:

```
if (i<=d && d<=N-1)
    S0(i,d);
```

and the second loop on `j` can be transformed as well (note that we do not keep a variable to store the value of `j`, but rather substitute it immediately).

The major advantage of having affine dates is the fact that the resulting program can be further optimized. We are going to illustrate these additional optimizations on the example program in Figure 6, and then we will show how the program transformations involved are strongly related to the problem of code generation from a polyhedral model of a program. We will then show, in the next section, how existing tools can be adapted to directly produce the optimized version.

Regarding the example of Figure 6, the first step is to replace constructs of the form `async if (...) S(...)` with `if (...) async S(...)`, because there is no need to create an activity that does nothing. All these initial modifications lead to the following program:

```
for (d in 0..(2*N-2))
  finish
    for (i in 0..(N-1)) {
      if (i<=d && d<=N-1)
        async S0(i,d);
```

[7] Note that the complexity in terms of the number of executions of individual instructions is always the same on both versions. Here we refer to the complexity of the associated control, i.e., the number of times guards and loop exit conditions are evaluated.

```
clocked finish {
  for (i in 0..(N-1)) {
    clocked async // i
      for (j in i..(N-1)) {
        S0(i,j); // i+j-i = j
        advance();
      }
    advance();
    clocked async // i+1
      for (j in 0..(i-1)) {
        S1(i,j); // i+1+j
        advance();
      }
  }
}
```

```
for (d in 0..(2*N-2))
  finish {
    for (i in 0..(N-1)) {
      async
        for (j in i..(N-1)) {
          if (d == j)
            S0(i,j);
        }

      async
        for (j in 0..(i-1)) {
          if (d == i+j+1)
            S1(i,j);
        }
    }
  }
```

**Fig. 6.** A polyhedral program with affine dates on the left, and the corresponding program after clock removal and before optimization on the right.

```
  if (i+1<=d && d<=2*i)
    async S1(i,d-i-1);
}
```

At this point, all remaining optimizations are made possible by the comparison of the various inequalities that apply to the individual instructions. Since our goal is to reduce the time taken by evaluating the guards, we are going to rearrange this code to eliminate useless guards and uselessly large bounds.

The first batch of useless evaluations of guards is caused by the `d<=N-1` condition, because at this point `d` is supposed to iterate from zero to `2*N-2`. This means that half of the values of `d` will simply fail to satisfy the condition. Eliminating these useless tests requires that the range of `d` is split into two sub-ranges, the first of which makes the condition trivially true, and the second which makes it false. The result appears in Figure 2.5. Range-splitting globally enlarges the code, but removes any occurrence of `S0` in the loop iterating over the second sub-range. Note also that condition `i+1<=d` around `S1` has become trivially true in the second loop, and is therefore also omitted.

The second set of unnecessary tests is caused by the remaining conditions, which in all cases are stricter than the surrounding loop bounds. The range of the first loop on `i` can be split into three sub-ranges, namely `0..(d-1)`, `d`, and `(d+1)..(N-1)`: the first leads to the bulk of the work, the second selects only `S0`, and the third leads to nothing. The result of bound adjustment appears in Figure 2.5. What was just done on upper bounds can now be done on lower bounds as well: the condition `d<=2*i` appears twice inside loops whose lower bound on `i` is zero, for any value of `d`. Therefore, the lower bound can be adjusted as well. The details are left to the reader.

```
for (d in 0..(N-1))
  finish
    for (i in 0..(N-1)) {
      if (i<=d)
        async S0(i,d);
      if (i+1<=d && d<=2*i)
        async S1(i,d-i-1);
    }
for (d in N..(2*N-2))
  finish
    for (i in 0..(N-1)) {
      if (d<=2*i)
        async S1(i,d-i-1);
    }
```

(a) After range splitting on `d`

```
for (d in 0..(N-1))
  finish {
    for (i in 0..(d-1)) {
      async S0(i,d);
      if (d<=2*i)
        async S1(i,d-i-1);
    }
    async S0(d,d);
  }
for (d in N..(2*N-2))
  finish
    for (i in 0..(N-1)) {
      if (d<=2*i)
        async S1(i,d-i-1);
    }
```

(b) After bound adjustments

**Fig. 7.** The program transformed from Fig. 6, after various further optimizations

## 3 Polyhedral Implementation and Optimized Control

The approach we have taken in the previous section consists in a succession of elementary transformations: wrapping a loop around the original code, placing guards around elementary statements, and adjusting iteration domains according to the guards. In contrast, in the polyhedral model, all these transformations can be represented in a uniform framework, and polyhedral operations can be used to manipulate the program. A polyhedral model of an instruction (a polyhedron, for short) is made up from two distinct parts: first, an ordered list of dimensions, and second a set of constraints (inequalities) on the values of the various dimensions. There are three types of dimensions: 1) syntactic dimensions, which are usually constants, 2) loop iterators, and 3) parallel constructs indicators, which are the abstract symbols `f(inish)` and `a(sync)`. Figure 8 displays the polyhedra corresponding to the instructions appearing in the original program of Figure 6. The left part of the figure shows an abstract syntax tree, which is convenient to read the various dimensions. The right part shows the polyhedra, using the notation of the `iscc` *polyhedral calculator*, part of the `barvinok` library [8]. Note that polyhedra can be parametrized (by `N` in our case), and that constant dimensions can be written literally, *i.e.*, `{[f,0,...]: ...}` is equivalent to `{[f,p_0,...]: p_0=0 and ...}`.

All manipulations necessary for the elimination of clocks can now be formulated as operations on polyhedra:

1. Introducing dates is performed by adding a dimension, at the very end of the list of dimensions since the date may depend on any of the enclosing loop counters. For instance, the definition of `S0` becomes:

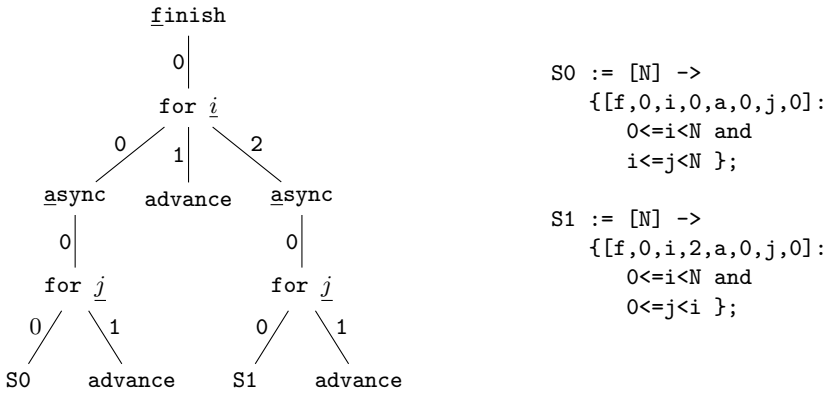


**Fig. 8.** An AST for the program on Figure 6, and the corresponding polyhedra

```
S0 := [N]->{[f,0,i,0,a,0,j,0,d]: 0<=i<N and i<=j<N and d=j};
```

2. Representing the whole program simply consists in computing the union of the individual instruction polyhedra:

   ```
   P := S0+S1;
   ```

   Here `P` represents the set of instances of `S0` and `S1`, in the order of the original program.[8]
3. Iterating on dates first is performed by changing the order of the dimensions. This is written as:

   ```
   U := {[f,p0,i,p1,a,p2,j,p3,d]->[d,f,p0,i,p1,a,p2,j,p3]}(P);
   ```

   This actually doesn't do anything, but is an important indication to the next step.
4. Producing the final code is performed by generating a program scanning the resulting polyhedron `U`. We use the CLooG algorithm [11], which produces a new loop nest with a loop scanning dates (`d`) first, and whose body contains various constructions (`finish`, `async`, loops, and instructions) in the order prescribed by the various other dimensions (the original CLooG algorithm had to be hacked to handle `finish` and `async`).

The final code (after trivial cosmetic post-processing) appears on Figure 9: CLooG has adjusted all loop bounds (even though it could have gone further). This code generation phase actually under-uses CLooG, which is able to apply a "scattering function" (taking its default value, the identity, in our case). The same result could be obtained by applying, e.g., some variation of Fourier-Motzkin elimination for bound adjustment [12]. However, reconstructing the structure would still need additional work. CLooG does both iteration domain computation *and* code generation.

[8] For this union operation to have any meaning, the dimension lists of the various must coincide; this is trivially achieved by padding with zeros. No modification is necessary in our example.

```
for (d in 0..(N - 1))
  finish
    for (i in 0..d) {
      async S0(i, d);
      if (d >= i + 1 && 2 * i >= d)
        async S1(i, d - i - 1);
    }
for (d in N..(2 * N - 2))
  finish
    for (i in (d - d / 2)..(N - 1))
      async S1(i, d - i - 1);
```

**Fig. 9.** The final result, produced by CLooG

## 4 Experimental Results

To evaluate the effect of eliminating clocks on execution time, we have used eight different polyhedral programs with affine dates. All these programs are parametrized by a number $N$ that determines the number of activities and the number of iterations of loops in various ways. Their execution is depicted on Figure 10 for $N = 6$: vertical lines represent activities, dots represent individual instruction executions, and horizontal dashed lines represent phases of execution.

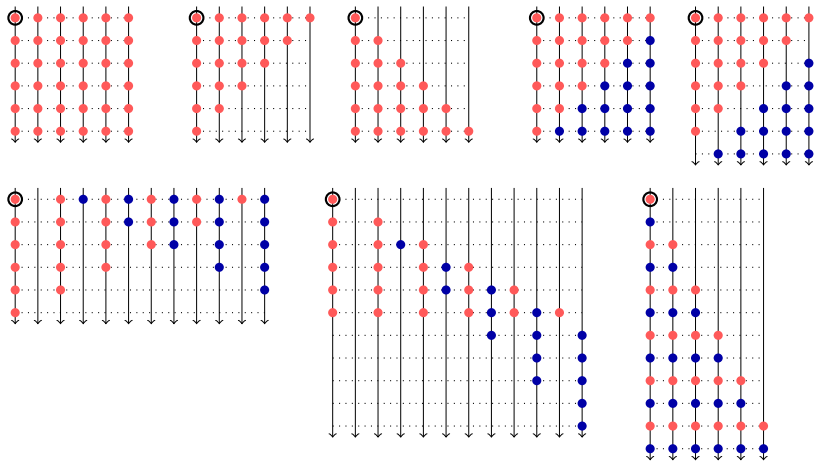

**Fig. 10.** Example iteration spaces, here for $N = 6$. All examples spawn $O(N)$ activities, last for $O(N)$ clock steps, and execute $O(N^2)$ instructions.

To compare the clocked and clockless versions of each program, we have measured their execution times (averaged over 20 executions). We have used a not-quite-recent X86-64 compatible AMD machine with 24 cores (two sockets of 12 cores). X10 programs were compiled with the official release of X10,

version 2.3.1, available from http://x10-lang.org/. All programs have been run with $N = 100$. Because the elimination of clocks affects only the control of the program, and not its actual work, we have varied the time taken by a single instruction execution (a call of the form $S_k(i, j)$ in all cases) the same way we did in Section 2.1: a single parameter $T$ controls how much time a single execution of any $S_k(i, j)$ takes. The goal of the experiment is therefore to measure the difference in execution times as a function of $T$.

The results are shown on Figure 11. Every graph shows the execution time of both versions. In all cases, the clockless version runs faster than the version with clocks. Rows of three graphs show the times of a given program for various values of $T$ (the workload): every graph displays ten evenly spaced values of $T$, with one order of magnitude variation from one graph to the next. The vertical scales are different from one graph to the other, but all scales are zero-based.

Since X10 is not the only language allowing finish/async programming, we have also conducted preliminary experiments with Habanero-Java [5] (version 1.3.1), with results similar to those presented here.

There are several lessons to learn from Figure 11. First, eliminating clocks always has a positive impact on execution time. This validates our intuition that clocks are expensive to manage. At least their use is more expensive than launching more activities (by a factor $O(N)$ in our case). We acknowledge that this is fairly dependent on implementation issues, but we also think that it will be easier to optimize activity creation rather than clock synchronization. Future implementations of X10 (and related languages, like Habanero and Chapel [2]) may change this situation.

Examination of the leftmost column of Figure 11 shows the relatively irregular behavior of programs using clocks with fine-grain instructions: it looks as if the frequent calls to `advance()` make the actual time difficult to predict, whereas clockless programs display a smoother, quasi-linear curve. Again, this heavily depends on the implementation of the activity scheduler, but it seems clear that clockless programs are "easier" to schedule over an arbitrary number of threads.

The third lesson learned is that, as expected, the difference between versions vanishes when the workload is reasonably large, because the time spent in control becomes negligible compared to the time spent on computation. What is less obvious is that this happens for values of $T$ around one million (which, on our machine, is about 1 millisecond). Considering the kind of programs we have used (basically loops over arrays, where every instruction accesses one element of one or more arrays), there is little chance that this workload is reached. This means that for fine-grain programs, the transformation is probably advantageous, providing significant speedup in most cases.

## 5 Related Work

There exists a large body of literature on barriers and clocks, their analysis, optimization and verification. Nearest to the subject of this paper is work on

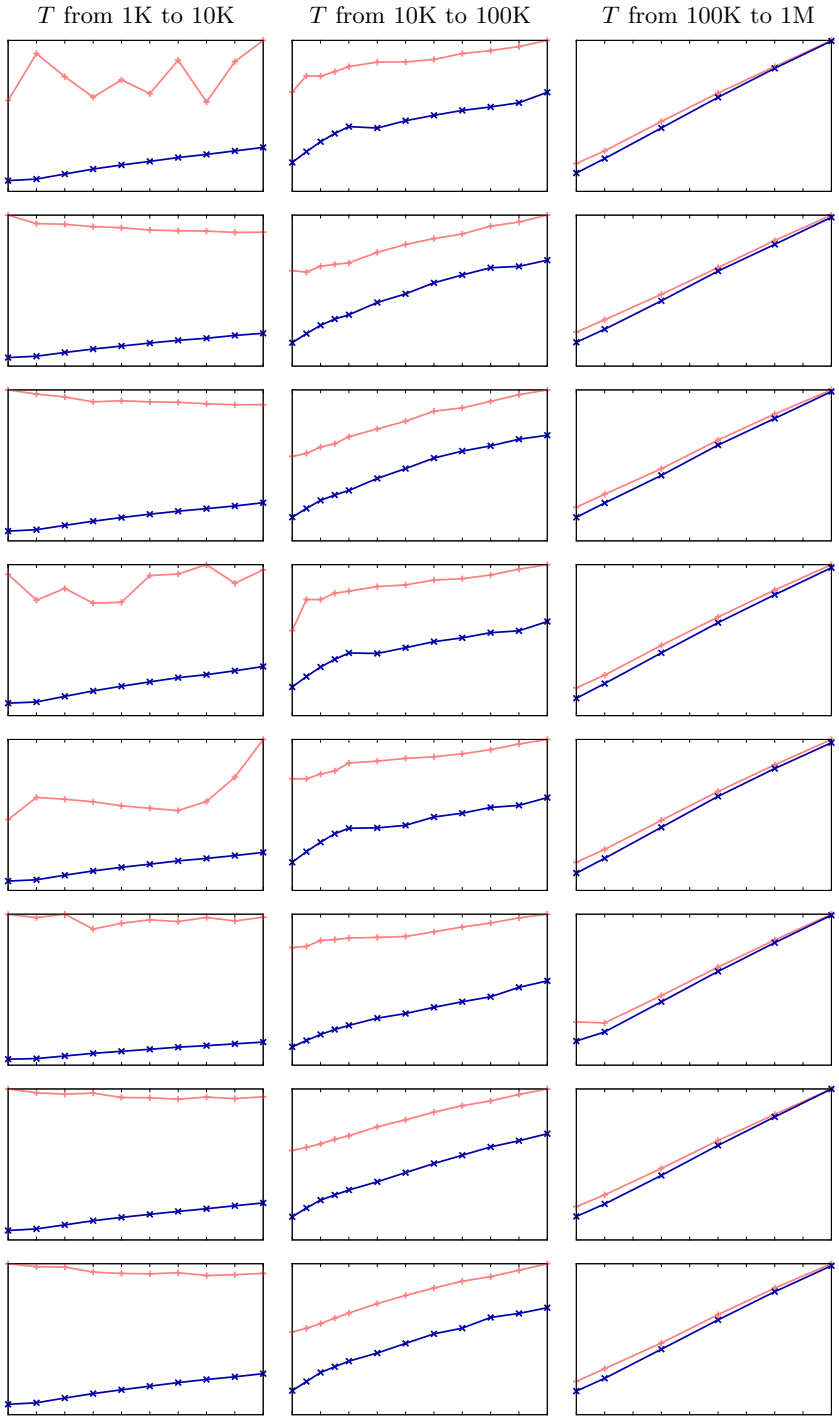


**Fig. 11.** Execution times for clocked (+) and clockless (×) versions, for various scales of workload. Vertical scales vary from graph to graph, but are all zero-based.

optimal barrier placement [13–15] and verification [16]. While apparently related to the present work, Chau-Weng Tseng paper [17] deals in fact with a completely different problem, namely how to distribute work among threads in order to minimize synchronization. Several authors have argued that barriers or clocks can be implemented more efficiently than task or activity creation, and have advocated algorithms for minimizing the number of tasks. To the best of our knowledge, the word *SPMDization* was coined by Padua and Paek in [18]. A recent discussion of the same idea is by Zaho et. al. [19] in which an algorithm, which amount to moving parallel loops outside sequential loops with barrier insertion is proposed. Our contention here is that moving in the opposite direction may be beneficial in some cases. Choosing between the two solutions depends on many factors: the target system, the compiler and runtime and the source program. For instance, if the target is hardware, where it is almost impossible to dynamically create activities and synchronization is cheap, using clocks might be the best solution. Our work shows that the situation is exactly the reverse for software.

## 6 Conclusion

When one has to generate a parallel program, either manually or automatically, one has to choose between two extreme program shapes: one a sequence of parallel constructs, the other a parallel composition of sequential threads. Obviously, these two extreme cases can be combined to produce many intermediate solutions.

In the first approach, it is usually possible to restrict synchronization to one barrier after each parallel block. This corresponds to the exclusive use of `async / finish` in X10, and is especially suitable for vector or data-parallelism. In the second approach, it is usually not possible, except in the case of embarassingly parallel programs, to construct independent threads. Residual dependences must be satisfied using clock or phasers [5]. This work shows that deciding which approach gives the best performance is not obvious, and must be approached experimentally. Our main contribution is a systematic method for converting a large class of clocked programs into unclocked ones. Our algorithms can easily be automated, thus simplifying the comparison process.

This paper has introduced a program transformation that acts on an explicitly parallel program, an unusual characteristic in the polyhedral framework. Such an ability opens up a large space of new potential optimizations, extending the scope of automatic parallelization. Taking the cost of synchronization primitives into account must also be extended and further generalized, to cover cases where implementations may have different semantics and/or relative overheads. Also, the cost of synchronization is only one part of the picture, and more work is needed to combine synchronization costs with more "traditional" transformation objectives in the polyhedral framework, like, e.g., temporal and spatial locality. Finally, we plan to investigate the use of parallel-to-parallel program transformations in dynamic optimization frameworks, where switching between various

versions of the same program can alleviate the variation of synchronization costs linked to changing runtime conditions.

## References


1. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., et al.: Titanium: A high-performance Java dialect. Concurrency Practice and Experience **10**(11-13) (1998) 825–836
2. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications **21**(3) (2007) 291–312
3. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum **17**(2) (August 1998) 1–31
4. Consortium, U., et al.: UPC language specifications. Lawrence Berkeley National Lab Tech Report LBNL–59208 (2005)
5. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-java: the new adventures of old X10. In: PPPJ '11, ACM (2011) 51–61
6. Feautrier, P., Lengauer, C.: The polyhedral model. In Padua, D., ed.: Encyclopedia of Parallel Programming. Springer (2011)
7. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array dataflow analysis for polyhedral X10 programs. In: PPoPP. (2013)
8. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. In: Algorithmica. (2007)
9. Lee, J., Padua, D.A., Midkiff, S.P.: Basic compiler algorithms for parallel programs. In: PPoPP '99, ACM (1999) 1–12
10. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In: ICS '96, ACM (1996) 278–285
11. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT'13, Juan-les-Pins (september 2004) 7–16
12. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming, ACM Press (April 1991) 39–50
13. Aiken, A., Gay, D.: Barrier inference. In: POPL'98. (1998) 342–354
14. Kamil, A., Yelick, K.: Concurrency analysis for parallel programs with textually aligned barriers. In: LCPC. (2005)
15. Darte, A., Schreiber, R.: A linear-time algorithm for optimal barrier placement. In: PPoPP '05, ACM (2005) 26–35
16. Vasudevan, N., Tardieu, O., Dolby, J., Edwards, S.A.: Compile-time analysis and specialization of clocks in concurrent programs. In: Compiler Construction. CC '09, Springer-Verlag (2009) 48–62
17. Tseng, C.W.: Compiler optimizations for eliminating barrier synchronization. In: PPoPP '95, ACM (1995) 144–155
18. Padua, D.A., Paek, Y.: Compiling for scalable multiprocessors with Polaris. Parallel Processing Letters **07**(04) (1997) 425–436
19. Zhao, J., Shirako, J., Nandivada, V.K., Sarkar, V.: Reducing task creation and termination overhead in explicitly parallel programs. In: PACT '10, ACM (2010) 169–180